



User-Defined Call Functions in the PeriProducer Environment

**Avaya Business Communications Manager
Release 6.0**

Document Status: **Standard**

Document Number: **P0603935**

Document Version: **3.7**

Date: **June 2010**

© 2010 Avaya Inc.
All Rights Reserved.

Notices

While reasonable efforts have been made to ensure that the information in this document is complete and accurate at the time of printing, Avaya assumes no liability for any errors. Avaya reserves the right to make changes and corrections to the information in this document without the obligation to notify any person or organization of such changes.

Documentation disclaimer

Avaya shall not be responsible for any modifications, additions, or deletions to the original published version of this documentation unless such modifications, additions, or deletions were performed by Avaya. End User agree to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation, to the extent made by End User.

Link disclaimer

Avaya is not responsible for the contents or reliability of any linked Web sites referenced within this site or documentation(s) provided by Avaya. Avaya is not responsible for the accuracy of any information, statement or content provided on these sites and does not necessarily endorse the products, services, or information described or offered within them. Avaya does not guarantee that these links will work all the time and has no control over the availability of the linked pages.

Warranty

Avaya provides a limited warranty on this product. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya's standard warranty language, as well as information regarding support for this product, while under warranty, is available to Avaya customers and other parties through the Avaya Support Web site: <http://www.avaya.com/support>

Please note that if you acquired the product from an authorized reseller, the warranty is provided to you by said reseller and not by Avaya.

Licenses

THE SOFTWARE LICENSE TERMS AVAILABLE ON THE AVAYA WEBSITE, [HTTP://SUPPORT.AVAYA.COM/LICENSEINFO/](http://support.avaya.com/licenseinfo/) ARE APPLICABLE TO ANYONE WHO DOWNLOADS, USES AND/OR INSTALLS AVAYA SOFTWARE, PURCHASED FROM AVAYA INC., ANY AVAYA AFFILIATE, OR AN AUTHORIZED AVAYA RESELLER (AS APPLICABLE) UNDER A COMMERCIAL AGREEMENT WITH AVAYA OR AN AUTHORIZED AVAYA RESELLER. UNLESS OTHERWISE AGREED TO BY AVAYA IN WRITING, AVAYA DOES NOT EXTEND THIS LICENSE IF THE SOFTWARE WAS OBTAINED FROM ANYONE OTHER THAN AVAYA, AN AVAYA AFFILIATE OR AN AVAYA AUTHORIZED RESELLER, AND AVAYA RESERVES THE RIGHT TO TAKE LEGAL ACTION AGAINST YOU AND ANYONE ELSE USING OR SELLING THE SOFTWARE WITHOUT A LICENSE. BY INSTALLING, DOWNLOADING OR USING THE SOFTWARE, OR AUTHORIZING OTHERS TO DO SO, YOU, ON BEHALF OF YOURSELF AND THE ENTITY FOR WHOM YOU ARE INSTALLING, DOWNLOADING OR USING THE SOFTWARE (HEREINAFTER REFERRED TO INTERCHANGEABLY AS "YOU" AND "END USER"), AGREE TO THESE TERMS AND CONDITIONS AND CREATE A BINDING CONTRACT BETWEEN YOU AND AVAYA INC. OR THE APPLICABLE AVAYA AFFILIATE ("AVAYA").

Copyright

Except where expressly stated otherwise, no use should be made of the Documentation(s) and Product(s) provided by Avaya. All content in this documentation(s) and the product(s) provided by Avaya including the selection, arrangement and design of the content is owned either by Avaya or its licensors and is protected by copyright and other intellectual property laws including the sui generis rights relating to the protection of databases. You may not modify, copy, reproduce, republish, upload, post, transmit or distribute in any way any content, in whole or in part, including any code and software. Unauthorized reproduction, transmission, dissemination, storage, and/or use without the express written consent of Avaya can be a criminal, as well as a civil offense under the applicable law.

Third Party Components

Certain software programs or portions thereof included in the Product may contain software distributed under third party agreements ("Third Party Components"), which may contain terms that expand or limit rights to use certain portions of the Product ("Third Party Terms"). Information regarding distributed Linux OS source code (for those Products that have distributed the Linux OS source code), and identifying the copyright holders of the Third Party Components and the Third Party Terms that apply to them is available on the Avaya Support Web site: <http://support.avaya.com/Copyright>.

Trademarks

The trademarks, logos and service marks ("Marks") displayed in this site, the documentation(s) and product(s) provided by Avaya are the registered or unregistered Marks of Avaya, its affiliates, or other third parties. Users are not permitted to use such Marks without prior written consent from Avaya or such third party which may own the Mark. Nothing contained in this site, the documentation(s) and product(s) should be construed as granting, by implication, estoppel, or otherwise, any license or right in and to the Marks without the express written permission of Avaya or the applicable third party. Avaya is a registered trademark of Avaya Inc. All non-Avaya trademarks are the property of their respective owners.

Downloading documents

For the most current versions of documentation, see the Avaya Support. Web site: <http://www.avaya.com/support>

Contact Avaya Support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Web site: <http://www.avaya.com/support>

Table of Contents

Preface.....	5
Scope	6
Intended Audience	6
How to Use This Manual	6
Organization of This Manual	7
Conventions Used in This Manual	7
Solaris and Windows 2000 Conventions	8
 Chapter 1 — Overview	 9
Overview	10
 Chapter 2 — Call Functions	 11
User Defined Call Functions	12
Calling Conventions	12
Accessing Arguments	14
Verifying Arguments	16
Signal Handling (Solaris Only).....	16
Memory Access	17
Error Processing	18
Function Prototype	18
Connecting to VMST	20
Compiling and Using Shared Libraries	22
Compiling Shared Libraries	22
Makecall on Solaris	22
Makecall on Windows 2000	23
Using Shared Libraries	24
Including Third Party Software	25
Debugging	28
Calldaemon	29

This page has been intentionally left blank.

Preface

Scope

This manual is intended as a guide to producing and integrating custom Call Functions for use in the PeriProducer environment. It should be used in conjunction with the *PeriProducer User's Guide*. It is not intended as a style guide or tutorial on how to program a custom Call Function.

Examples contained in this manual are for illustration purposes only. The techniques and styles presented in the examples should not be considered as necessary or appropriate for any specific application.

Intended Audience

This manual is intended for those who will be creating custom Call Functions and integrating them into a PeriProducer environment. The reader should be familiar with programming in C and using Solaris system compilers and utilities. The reader should also be familiar with the Avaya Media Processing Server (MPS) Series system and have a working knowledge of PeriProducer. It is assumed that the reader has completed MPS Series system and PeriProducer training courses conducted by Avaya.

How to Use This Manual

This manual uses many standard terms relating to computer system and software application functions. However, it contains some terminology that can only be explained in the context of the MPS Series system. Refer to the *Glossary of MPS Terminology* for definitions of MPS Series specific terms.

Initially, you should read this manual at least once, from start to finish. Later, you can use the Table of Contents to locate topics of interest for reference and review.

If you are reading this document online, use the cross-reference links (shown in [blue](#)) to quickly locate related topics. <LEFT> click once with your mouse while positioned with your cursor over the cross-reference link. Click on any point in a Table of Contents entry to move to that topic. Click on the page number of any Index entry to access that topic page.

To familiarize yourself with various specialized textual references within the manual, see [Conventions Used in This Manual](#) on page 7.



Periphonics is now part of Avaya. The name Periphonics, and variations thereof, appear in this manual only where it is referred to in a product. (For examples, a PeriProducer application, the PERImps package, the **perirev** command, and so on.)

Organization of This Manual

This manual is organized into separate chapters explaining the features and functions of PeriProducer.

Chapter 1 — Overview

This chapter provides an introduction to creating User-Defined Call Functions for use in a PeriProducer environment.

Chapter 2 — Call Functions

This chapter provides instructions that apply to integrating User-Defined Call Functions into the PeriProducer environment.

Conventions Used in This Manual

This manual uses different fonts and symbols to differentiate between document elements and types of information. These conventions are summarized in the following table.

Conventions Used in This Manual Sheet 1 of 2

Notation	Description
Normal text	Normal text font is used for most of the document.
<i>important term</i>	The Italics font is used to introduce new terms, to highlight meaningful words or phrases, or to distinguish specific terms from nearby text.
system command	This font indicates a system command and/or its arguments. Such keywords are to be entered exactly as shown (i.e., users are not to fill in their own values).
command, condition and alarm	Command, Condition and Alarm references appear on the screen in magenta text and reference the <i>Command Reference Manual</i> , the <i>PeriProducer User's Guide</i> , or the <i>Alarm Reference Manual</i> , respectively. Refer to these documents for detailed information about Commands , Conditions , and Alarms .
file name / directory	This font is used for highlighting the names of disk directories, files, and extensions for file names. It is also used to show displays on text-based screens (for example, to show the contents of a file.)
on-screen field	This font is used for field labels, on-screen menu buttons, and action buttons.
<KEY NAME>	A term that appears within angled brackets denotes a terminal keyboard key, a telephone keypad button, or a system mouse button.
<i>Book Reference</i>	This font indicates the names of other publications referenced within the document.

Conventions Used in This Manual Sheet 2 of 2

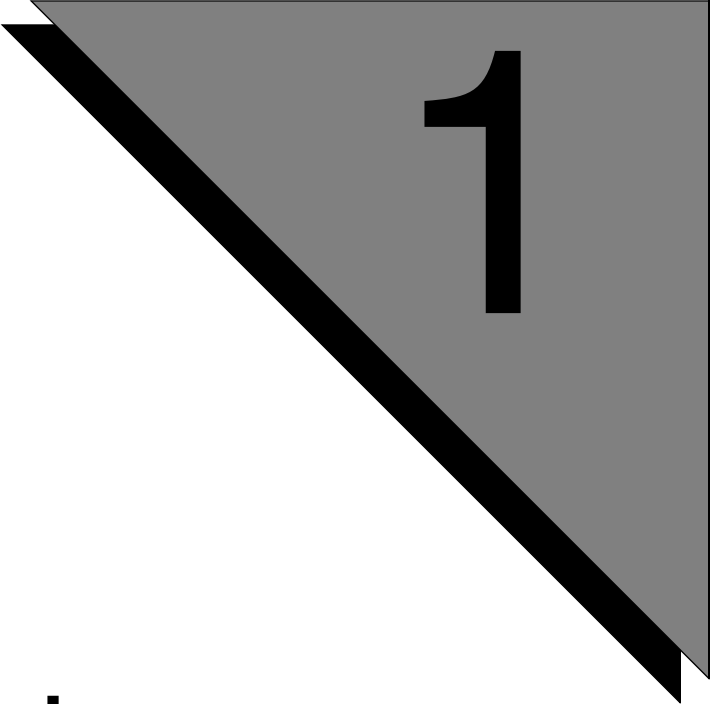
Notation	Description
cross reference	A cross reference is shown on the screen in blue . Click on the cross reference to access the referenced location. A cross reference that refers to a section name accesses the first page of that section.
	The Note icon identifies notes, important facts, and other keys to understanding.
	The Caution icon identifies procedures or events that require special attention. The icon indicates a warning that serious problems may arise if the stated instructions are improperly followed.
	The flying Window icon identifies procedures or events that apply to the Windows 2000 operating system only. ¹
	The Solaris icon identifies procedures or events that apply to the Solaris operating system only. ²

1. Windows 2000 and the flying Window logo are either trademarks or registered trademarks of the Microsoft Corporation.
2. Solaris® is a registered trademark of The Open Group in the U.S. and other countries.

Solaris and Windows 2000 Conventions

This manual depicts examples (command line syntax, configuration files, and screen shots) in Solaris format. In certain instances Windows 2000 specific commands, procedures, or screen shots are shown where required. The following table lists examples of general operating system conventions to keep in mind when using this manual with either the Solaris or Windows 2000 operating system.

	Solaris	Windows 2000
Environment	<code>\$MPSHOME</code>	<code>%MPSHOME%</code>
Paths	<code>\$MPSHOME/common/etc</code>	<code>%MPSHOME%\common\etc</code>
Command	<code><command> &</code>	<code>start /b <command></code>



1

Overview

This chapter covers:

1. Overview

Overview

The implementation of PeriProducer in the ASE environment is user extensible. If a desired programming operation cannot be effectively accomplished with PeriProducer blocks and/or with the set of predefined Call Functions (available from within the *System* block), application developers may create their own Call Functions using the C-language to supplement the base language constructs.



For additional information on PeriProducer, see the *PeriProducer User's Guide*.

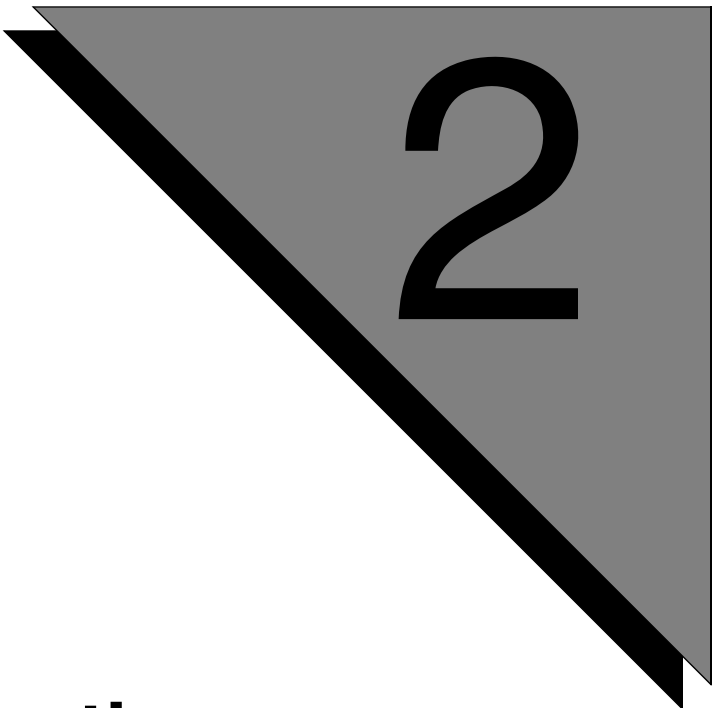
User-defined Call Functions are implemented via *shared libraries*. Accessing user-defined Call Functions bears some similarity to the use of PeriProducer subprograms through program linking. To convey appropriate status information to the main program from the Call Function, users may also create their own status/exception conditions.



As a general recommendation, application developers should attempt to implement programs using only PeriProducer, before concluding that a given program task requires a custom-designed Call Function.

When creating a user-defined function, the following steps must be taken by the application developer to ensure compatibility with the PeriProducer environment:

1. A Call Function is written as a C-language function, following the prescribed conventions for coding. See [Calling Conventions](#) on page 12.
2. The function is compiled and placed into a shared library, using the provided shell-script procedure **makecall**.
3. The name of the Call Function library is specified in the Application Configuration window as the Call Function Library Name (under the Execution Options menu). Multiple Call Function library names can be specified.



2

Call Functions

This chapter covers:

1. User Defined Call Functions
2. Compiling and Using Shared Libraries
3. Debugging
4. Calldaemon

User Defined Call Functions

User Defined Call Functions are necessary to:

- implement functionality otherwise unavailable in PeriProducer (binary byte manipulation, access to system resources, and so on),
- interface with third party software, and
- implement functionality that could be achieved in PeriProducer but with higher overhead.



Creating User Defined Call Functions should be avoided, unless it is absolutely necessary.

The template for:

- a Solaris User CALL Function is provided in the file:
`$ASEHOME/etc/new-function.c`
- a Windows 2000 User CALL Function is provided in the file:
`%ASEHOME%\etc\new-function.c`

For both Solaris and Windows 2000, the `new-function.c` file supports C and C++. The file has to be renamed to `new-function.cpp` before using a C++ compiler.

Calling Conventions

The following conventions differ somewhat from previous releases; however, full backward compatibility and complete functionality of legacy call functions are ensured.

A call function is called from a PeriProducer application by its name. Usually one or several folders/datacards are passed to the function as arguments. Arguments can also be represented by constants or indexed variables (folders/datacards). For example,

Call "my-function" Using Folder1, "Hello world", Item(2), Folder2

At run time, VENGINE invokes the function and passes arguments to it for processing. The function modifies one or more of the passed folders or datacards (not constants) and must return 0 upon success. If the function needs to pass an error condition back to the application, it must do so as described in "Error Processing" on page 18.

When it is required to pass more than six parameters to a call function, a Custom Container needs to be created (the System block is limited to only six parameters). The Custom Container must contain only:

- One Entry block,
- One Exit block, and
- One System block.

The System block should call the desired call function (via `Call an External Function`), and pass the Custom Container's Parameter folder as its only parameter. This folder should have one entry for each parameter being passed to the call function. There is no limit to the number of parameters in a folder.

It is important to fill in the description field for each member of the parameter folder (the description becomes the label of the corresponding parameter and is displayed when the Custom Container is opened).

After the Custom Container has been created, add it to the Tool Kit. It now can be used like any other Tool Kit block.



If you are going to use calldaemon for application debugging (see [Calldaemon on page 29](#)), you cannot pass complex data structures (for example, arrays, folders, custom containers, and so on) to the call function. ONLY LITERAL VALUES AND INDIVIDUAL DATACARDS ARE ALLOWED.



For more information about adding parameters and using Custom Containers, see the *PeriProducer User's Guide*.

Accessing Arguments

A call function must be named `cf_functionname`, in the C-language (the function name is the name as it is used by PeriProducer with all hyphens changed to underscores). For example, the function "*new-function*" should be named in `cf_new_function` in C and have a prototype:

```
int cf_new_function(envir *env);
```

The pointer `env` is defined in the file `$ASEHOME/include/vram.h` and is used to provide access to VENGINE's runtime environment for VENGINE-supplied functions. Ordinarily, the pointer `env` is not directly used by a call function.

Each argument passed to the call function will be represented by a pointer to an internal structure (`struct Symbol *sym`), and a corresponding index (`indsym`). `Indsym` has a value of 0, if the argument is **not** indexed (for example, passing folder A generates `indsym=0`, while `A(1)` generates `indsym=1`). Pairs (`sym`, `indsym`) are usually used as a unit to access the argument's values. VENGINE provides the function:

```
int vr_callargs(envir *env, struct Symbol
*symaddr[], int indaddr[])
```

to retrieve the number of arguments passed to the call function, and retrieve corresponding argument pairs. For example, the pair corresponding to the first argument is defined as:

```
(sym = symaddr[0], indsym = indaddr[0])
```

The data value corresponding to each argument pair (`sym`, `indsym`) is usually represented internally in ASCII format, and may be referred to by the pointer `char * ivalue;`. This value can be obtained by calling the function:

```
ivalue = vr_ivalue(env, sym, indsym)
```



If an argument's index changes, the pointer `ivalue` has to be recalculated. Incrementing `sym` is **not** equivalent to increasing `indsym`. For example, you can pass an array folder to a function, and then access its elements by changing the index, as follows:

```
for (indsym = 1; indsym < 48; indsym ++)  
    ivalue = vr_ivalue(env, sum, indsym);
```



Values are **not** null-terminated and cannot be manipulated as strings; therefore, the following code can cause a core dump (in Solaris), or memory problems that are difficult to diagnose:

```
printf("IT'S INVALID CODE %s\n", ivalue)
```

The number of bytes that correspond to the current argument (starting from `ivalue`), may be determined by the function:

```
ilength = vr_itemlen(env,sym)
```

The length of the argument data does not depend on its index. Given `ilength`, it is possible to manipulate the value. For example, to set it to all spaces:

```
(void)memcpy(ivalue, ' ', ilength)
```

When an argument (`sym`, `indsym`) corresponds to a folder, it is possible to access its subfolders by incrementing the pointer `sym`. For example, `(sym+1)` points to the first subfolder, `(sym+2)` points to the second subfolder, and so on.

Because of proprietary internal data representation in VENGINE, you cannot move data directly to a location referenced by the `vr_ivalue()` pointer. The system provides the following functions to move data into an argument, or to retrieve the value of an argument:

Functions for Accessing Arguments

Function	Description
<code>vr_itemlen(env,sym)</code>	Returns the item's length.
<code>vr_moveia(env,lv,sym,ind)</code>	To move lv (<i>long</i>) to an argument.
<code>vr_moveda(env,dv,sym,ind)</code>	To move dv (double precision) to an argument.
<code>vr_movesa(env,sv,sym,ind)</code>	To move sv (null-terminated string) to an argument.
<code>vr_movei(env,symf,indf,symt,indt)</code>	To move data from an symf argument to an symt argument.
<code>vr_symtola(env,sym,ind)</code>	To retrieve the value of an argument as a <i>long</i> .
<code>vr_symtoda(env,sym,ind)</code>	To retrieve the value of an argument as a double precision number.
<code>vr_asymtosa(env,sym,ind)</code>	To retrieve the value of an argument as a null-terminated string. The string will be automatically deallocated at the end of the current block (paragraph); therefore, it cannot be retained in the call function between calls, and should not be deallocated explicitly.

Verifying Arguments

VENGINE cannot completely check the validity of all arguments passed to a call function. For example, it cannot determine when passing a *constant* instead of a *data card* or *folder* is an error (when the argument needs to be modified by the function). VENGINE does verify all indexing.

To avoid errors at processing time, the verify the call function's arguments on entry. In particular, the number of arguments returned by the call:

```
nargs = vr_callargs(env, &symaddr, &indaddr);
```

Argument and application properties can also be checked with the following functions:

Functions and Macros for Checking Properties

Function/Macro	Return Value
vr_constitem(env,sym)	Non-zero value if the corresponding argument is a constant (cannot be changed).
vr_shareditem(env,sym)	Non-zero value if the corresponding argument is located in Shared Memory.
vr_itemname(env,sym)	Name of the item.
vr_paragname(env,0)	Current block (paragraph) name.
SIGNED(sym)	Non-zero value if the corresponding argument is signed (the VRAM picture clause contains S).
ARRAYITEM(sym)	Non-zero value if the corresponding argument can be indexed

If the function determines an error, it should return a condition to the application.

Signal Handling (Solaris Only)

A call function must not perform any operation that will cause a signal to be delivered to the VENGINE process, unless the call function installs a handler for this signal.



A call function must not assume that signal disposition is preserved after returning from the call function or between subsequent calls to the same or different call functions. If the call function changes the disposition of a signal it is good practice to restore the signal disposition before returning from the call function.

After returning from a call function, VENGINE restores signal disposition for the following signals:

SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGALRM, SIGTERM, SIGUSR1, SIGWINCH, SIGTTOU.

After returning from a call function, VENGINE cancels any previously set alarm clock by calling `alarm(0)`. VENGINE also resets signal disposition for `SIGALRM`, so after a return, a call function cannot rely on an alarm clock set inside the function.

Memory Access

Do not pass shared memory arguments to a call function for update. The system releases all shared memory locks before entering the call function. Modifying these arguments from the function can create "windows" when the memory state is undefined.



A call function must not allocate memory unless it is deallocated by the same or subsequent function calls. VENGINE is not responsible for memory management in user-defined call functions.

A call function must not try to deallocate memory returned by calls to VENGINE defined functions (for example, `vr_ivalue`, `vr_asymtosa`, `vr_itemname`, and so on.)

Error Processing

A call function must return 0 upon success. The call function must generate a condition if there is an error to the application.



To set correct error processing procedures, pass a return code from the `pr_error(envir *env, char *cond, char *fmt, ...)` function to VENGINE, as follows:

```
return vr_prerror(env, "invreq", "CALL",  
    "An attempt to modify a constant %s item\n",  
    vr_itemname(env, sym));
```

The second argument of the `vr_prerror()` function can be any condition, which can be handled or ignored by the application. The first and the third arguments have to be coded as shown, the fourth and consequent arguments define text to be printed if VENGINE is running with the command line option **-a** (condition messages are displayed).

Function Prototype

The function `EXPORT_CALL_FUNC` prints all of its argument values and sets non-constant arguments to spaces. This macro addresses function requirements for both the C and C++ languages on Solaris and Windows 2000.

`CALLVCHECK(env)` assures that the function has been recompiled to match the current version of VENGINE.

Printing text from a call function and explicitly manipulating an argument's data space is shown for demonstration purposes only. Usually, call functions do not print to `STDIO` and should use functions to access data. See [Functions for Accessing Arguments](#) on page 15.

Sample Code:

```
#include <vram.h>

/*
 * CALL "new-function" Using arg1, arg2, ...
 *
 * The function prints items values and blanks nonconstants
 */
EXPORT_CALL_FUNC
cf_new_function(envir *env)      /* Note the function name agreement */
{
    struct Symbol**symaddr,*sym; /* Array of pointers to args */
    int *indaddr,indsym;         /* Array of arg indexes */
    char *dataddr;               /* Arguments' Data space */
    int nargs, datalen, k;       /* Number of args and data length */
/*
 * Get arguments. Note that indexing is checked by upper level func
 */
    CALLVCHECK(env);             /* Check library version */

    nargs=vr_callargs(env,&symaddr,&indaddr);
    if (nargs==0)
        return pr_error(env,"abend","CALL","Function needs an argument");

/*
 * Handle each argument
 */
    for (k = 0;k <nargs;k++) {
        sym = symaddr[k];        /* Get next argument and its index */
        indsym = indaddr[k];
        datalen = vr_itemlen(env, sym);
        (void)printf("Item'%s':value'%d',length %d\n",
                     vr_itemname(env,sym),vr_asymtosa(env,sym,indsym),
                     datalen);

        if (!vr_constitem(env,sym)){
            dataddr = vr_ivalue(env,sym,indsym);
            (void)memset(dataddr, ' ', datalen);
            vr_dwatch(env,sym,indsym);/* Inform the debugger about an item
                                         change */
        }
    }
    return 0;
}
```

Connecting to VMST

Usually call functions do not deal with the issue of connectivity to VMST, leaving this to VENGINE. However, if the function blocks for a period of time on a connection to a third party host, the call function could be required to monitor the VMST connection as well. The function could also determine phone line connectivity and act accordingly.

To ensure that user call functions that read from sockets, pipes, ttys, and so on, comply with the existing paradigm (for example, respond to AMU management requests and accept asynchronous messages arriving from VMST), they have to follow these rules:

- Before executing the `read()` function call, the availability of data has to be determined by the `select()` call. The standard application's socket for communicating with VMST must be included in the `select` read set.
- If data arrives from VMST, it should be read by the `vr_readmsg()` function. This function will process VVPMessages and store them for future use by the application. If the function returns a non-zero return code, the User call function should return immediately. Note the functions `vr_linkdown(env)` and `vr_connected(env)` may be used, after the arrival of the VVPMessage, to check the state of the connection to VMST, and the state of the phone call.

Functions for Checking Connectivity

Function	Returns
<code>vr_connected(env)</code>	Non-zero value if the phone call is connected (the application has received condition conn, but has not received disc).
<code>vr_linkdown(env)</code>	Non-zero value if the connection to VMST is lost (all communications with the outside world ceased).
<code>VVPSOCKET(env)</code>	The socket number corresponding to the VMST connection.
<code>vr_readmsg(env)</code>	0 if the call function can continue processing, otherwise it should return the value to the Calling Function immediately.

Sample Code:

```
UNLOCKSM(env);
if ( (socket = VVPSOCKET(env)) >= 0 ) {
    if ( socket > maxfd )
        maxfd = socket;
    FD_SET(socket, &readfds);
}

if ( select(maxfd + 1, &readfds, ...) ) {
    if ( socket >= 0 && FD_ISSET(socket, &readfds) ) {
        /*
         * There is a message from VMST.
         * Process it and return to the application
         * in the case of an emergency.
         */
        if ( (exit_code = vr_readmsg(env)) != 0 ||
            !vr_connected(env) )
            return exit_code;
    }

    /*
     * Your code to process call function specific
     * messages.
     */
    ...
}
```



This type of processing is complicated and should be avoided when possible.

Compiling and Using Shared Libraries

Compiling Shared Libraries

Applications Developers must compile call functions into shared libraries so that the call functions are available to VENGINE at runtime.



The library **must** be recompiled with each new ASE release. Avaya recommends that application developers recompile the library each time a new VENGINE is installed.

The macro `CALLVCHECK (env)` guarantees release compatibility by generating an error when the library needs to be recompiled, see [Function Prototype on page 18](#). Unfortunately, legacy call functions do not use the macro and should be modified to include it, or other means should be taken to ensure compatibility.

Makecall on Solaris

Use the **makecall** script to compile one or more functions into a shared library. The **makecall** script compiles and stores a C-language module for use as a call function, as follows:

makecall [-C *libname.a*] [-d] [-u *cc-opts*] [-I *ld-opts*] *C-filename(s)*

- **-C** *libname.a* optionally specifies the name of a user-defined library for the object code. Full or relative path names may be given. The default is: `$MPSHOME/lib/libcall.a`



For the **-C** option only a static library of the form *libxxxx.a* is allowed.

- **-d** optionally specifies that this C-function will support debugging.
- **-u** specifies C-compiler options to use while compiling the function.
- **-I** specifies the link editor options to use.
- *C-filename(s)* specifies the name(s) of the C-language file(s) to be compiled. They must reside in the current working directory.

The following is an example of a required use of quotation marks around the **-u** and **-I** options (due to the slashes in the path):

```
makecall -C libnew.a -u '-I/home/peri/xyz.h' my.c
```

Makecall on Windows 2000

On Windows 2000, the **makecall** script requires you to specify the library name with the `.dll` extension. The C-filename(s) must be specified without an extension. It is the responsibility of the user to provide correct arguments. For example:

```
makecall -C newlib.dll newfunction
```

The default is `%ASEHOME%\libcall.dll`. The other options are the same for Windows 2000 as for Solaris, see [Makecall on Solaris on page 22](#).

Unlike the Solaris environment where a new function is **added** to the library if it already exists, the script on Windows 2000 will create a library which contains only the functions being compiled.



To invoke the **makecall.cmd** script from inside another script (or any script from inside of another), the command **call** must be used:

```
call makecall -C newlib.dll newfunction
```

Using Shared Libraries

At runtime, a shared library is made available to VENGINE by its command line option, **-C libname**:

```
venge -C libname1.a -C libname2.a ...
```

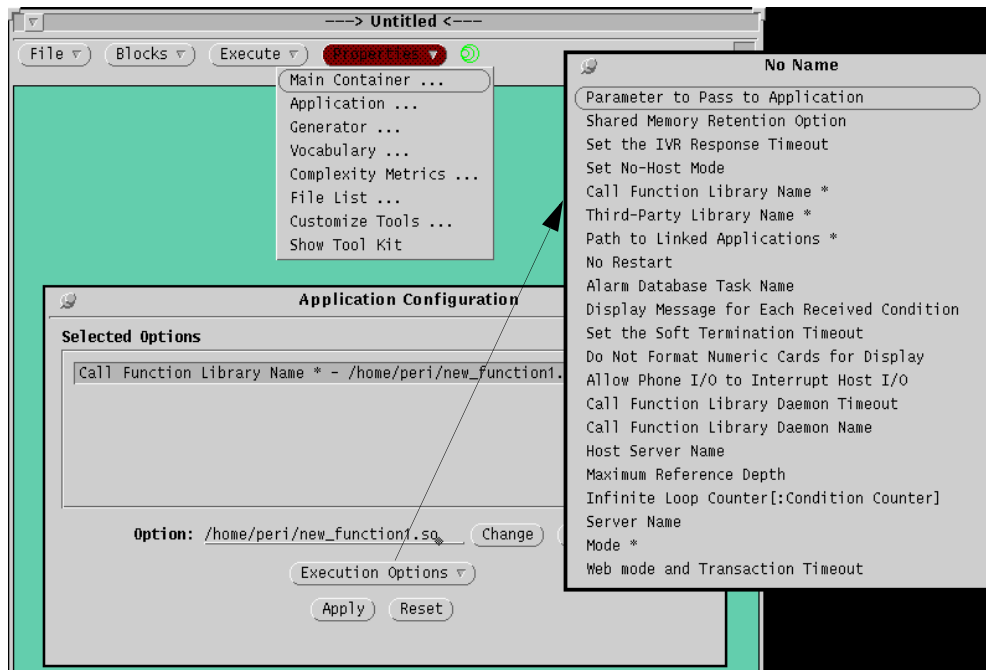
Multiple options for **-C** may be specified. The library name can include the full path.



On Windows 2000, shared libraries must have all references resolved. Therefore, there is no need to rebuild calldaemon, and the script **makecalldaemon** does not exist.

Shared libraries may also be included in the PeriProducer application by performing the following:

1. In the Main Container, select **Application** from the **Properties** menu.
2. In the Application Configuration window, select **Call Function Library Name** from the **Execution Options** menu
3. Specify the full path and filename of the shared library and click on **Change**.



Repeat this procedure to add as many Call Function Libraries as are required.

4. Recompile the application.

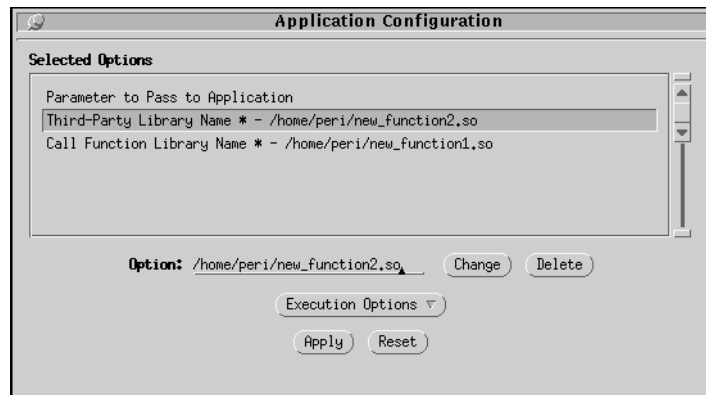


If an application is configured by **appman**, the **appman** function copies all user-defined call function libraries into `$MPSHOME/componentNAME/apps/lib`. The path to this location must be specified to VENGINE. **appman** adjusts the VENGINE command line arguments to indicate the path to the location. Third party shared libraries are not copied by **appman**.

Including Third Party Software

To include third party libraries:

1. In the Main Container, select **Application** from the **Properties** menu.
2. In the Application Configuration window, select **Third Party Library Name** from the **Execution Options** menu
3. Specify the full path and filename of the shared library and click on **Change**.



Each Third-Party Library added creates a **-B** VENGINE command line option.



Third party shared libraries are not copied to the systems where the applications are deployed. Third party libraries must already be installed on the specified systems.

In a Windows 2000 environment, you cannot build a shared library that has unresolved references. If the call function uses third party software, the application developer must modify the **makecall** script, or to invoke its options in a way that all references are included into the library and are resolved.

In a Solaris environment, you cannot build a shared library that has unresolved references. For example, the C function can call a third party function that is available in a separate object file, or is supplied as a static or dynamic library. A call function can also call a standard C function, such as `msgget()` (to process a message queue).

All such references must be resolved at runtime by the Solaris linker program. Because VENGINE is linked statically, for each reference not resolved by a command line option, the Solaris linker program will search VENGINE's Name Space to try to resolve it. For example, a reference from the call function `printf()` is resolved since the function resides in VENGINE. However, references to standard C functions may fail because VENGINE itself never referenced them; hence, they have never been included into VENGINE (for example, `msgget()`).

To address this situation such as when a reference exists in another shared library, but not in VENGINE), VENGINE provides the command line option **-B shlib**. The full path to such a shared library has to be supplied to VENGINE. For example, to provide access to a standard C library, specify:

```
vengine -B /usr/lib/libs.so ...
```

Multiple options for **-B** may be specified.

The situation is more complicated if the call function needs to use Third Party functions supplied in object files or in static libraries. In this case, there may be the need to rebuild VENGINE to include missed references in its body. The **makevengine** script allows VENGINE to include such references. It has following arguments:

```
makevengine [ -o newname ] [ link_args ]
```

- **-o** specifies a new name. The default is: `$ASEHOME/bin/vengine`.
- *link_args* pass arguments to the link editor. This can be a reference to third party libraries, object files, and so on.

Linking is a complicated process which invokes each module only once as defined on the **makeengine** command line. Some modules may cause circular references which may need to be resolved by including the same object multiple times. Refer to standard Solaris documentation for details on the link editor **ld**. In particular, you can use the link editor's option **-u** to force the link editor to include particular references in the new VENGINE.

If some references are left unresolved at runtime, it can result in the condition **pgid** to the application. **pgid** can display a misleading message:

```
Call Function is not found.
```

This is because at runtime there is no way to determine which external name (the function's name or one of its references) was not found. The following Solaris command can be used to examine all external references in the library:

```
nm libcall.so | grep UNDEF
```

Debugging

Simple debugging is made available through VENGINE's built-in debugger. It can give notification each time a specified variable is modified. If the call function uses functions, such as `vr_moveda`, `vr_move1a`, and so on, to modify its arguments, the notification will be done automatically (if the `watch` variable is enabled). However, if the function accesses a data area directly by using the `ivalue` pointer, it has to notify the debugger by calling the `vr_dwatch(env, sym, indsym)` function. The following example sets a value to spaces and notify the built-in debugger:

```
(void)memcpy(ivalue, ' ', ilength);  
vr_dwatch(env, sym, indsym);
```

To use the system debugger (**dbx** or **workshop** on Solaris, **msdev** on Windows 2000), the call function has to be recompiled with debugging option **-d**.

Because references to the call function are resolved by the link editor only at the time of invocation, there is no way to set a break point in the user call function. To address this problem, VENGINE provides the artificial break point `vr_bpfunc()` function, which creates a window between the instant when the function is brought into VENGINE's data space and the time when it is actually called.

Use the following to set a break point in the `vr_bpfunc` function:

```
stop in vr_bpfunc
```

When execution stops at this break point, it is possible to set a break point to the function being debugged, as follows:

```
stop in cf_new_function
```

and then proceed with debugging.



If the break point is set to `vr_bpfunc`, execution stops each time **any** user call function is invoked (not only the function being debugged).

Calldaemon

You can isolate a user call function from the application process space. Use this process while debugging and, in some cases, while working in a production environment. If you do not use this option, the application can crash if a function crashes (for example, due to a User or Third Party code malfunction).

If the application is started with the VENGINE option **-E** time (**-E** uses a default timeout of 60 seconds), it creates a child. The child is the same for all call functions. That is, the daemon runs as a separate process, one per VENGINE for all call functions, and all function calls.

If the function crashes, it does not affect the application. VENGINE receives an **abend** condition and automatically restarts the daemon. VENGINE also times the call function execution.

If the call function does not return to the application within the specified time (for example, due to bugs in the user call function, the host not responding, and so on), the application considers the daemon dead and terminates it. In this case, the system produces the core file `./callcore` to help the programmer analyze the reason for failure. VENGINE will receive an **abend** condition. To disable the timer, use: **-E 0**.

Before using the calldaemon in a production environment, consider the associated overhead versus the probability of crashes.

If the call function uses Third Party Software, the calldaemon may need to be rebuilt (in the same way as VENGINE is rebuilt by **makeengine**).



On Windows 2000, shared libraries must have all references resolved; therefore, there is no need to rebuild calldaemon, and the script **makecalldaemon** does not exist.



If you are going to use calldaemon for application debugging (see [Calldaemon on page 29](#)), you cannot pass complex data structures (for example, arrays, folders, custom containers, and so on) to the call function. **ONLY LITERAL VALUES AND INDIVIDUAL DATACARDS ARE ALLOWED.**

makecalldaemon creates a daemon that can be used to execute a call function.

makecalldaemon [*link_options*] [**-o** *newname*]

- *link_options* specifies the linking options and path in which to place the daemon. The daemon is placed into \$ASEHOME/bin if the name does not contain the path.
- **-o newname** optionally specifies the name of the daemon ('calldaemon' is the default).

The following illustrates how to use this command to create a daemon called *newdaemon* in the current directory. It includes a third party object file called *def.o*, and library *abc* ("labc") from the directory */user/newlibs/*:

```
makecalldaemon -e ./newdaemon def.o -L /user/newlibs -labc
```

If the name is changed by the **-o** option, it has to be passed to VENGINE at runtime, by the **-e** option.